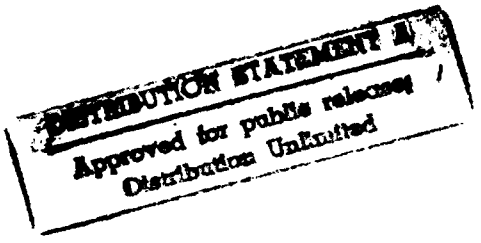
Eager Combining: A Coherency Protocol for Increasing Effective Network and Memory Bandwidth in Shared-Memory Multiprocessors

Ricardo Bianchini and Thomas J. LeBlanc

Technical Report 485
January 1994



UNIVERSITY OF
ROCHESTER
COMPUTER SCIENCE

Eager Combining: A Coherency Protocol for Increasing Effective Network and Memory Bandwidth in Shared-Memory Multiprocessors

Ricardo Bianchini and Thomas J. LeBlanc

ricardo@cs.rochester.edu, leblanc@cs.rochester.edu

The University of Rochester
Computer Science Department
Rochester, New York 14627

Technical Report 485

January 1994

One common cause of poor performance in large-scale shared-memory multiprocessors is limited memory or interconnection network bandwidth. Even well-designed machines can exhibit bandwidth limitations when a program issues an excessive number of remote memory accesses or when remote accesses are distributed non-uniformly. While techniques for improving locality of reference are often successful at reducing the number of remote references, a non-uniform distribution of references may still result, which can cause contention both in the interconnection network and at remote memories.

Producer/consumer data, where one processor (the producer) writes data that many other processors (the consumers) must read, is a common sharing pattern in parallel programs that generates a non-uniform distribution of references. In this paper we quantify the performance impact of producer/consumer sharing as a function of memory and network bandwidth, and argue that the contention caused by this form of sharing can severely impact performance on large-scale machines. We then propose a new coherency protocol, called *eager combining*, which is designed to alleviate this contention. The protocol replicates the producer's data among multiple memory modules, thereby effectively increasing both the memory and network bandwidth of the producer, and dramatically decreasing the remote access latency of consumers. We compare eager combining to other techniques for reducing or eliminating contention, and use execution-driven simulation of parallel programs on a large-scale multiprocessor to show that eager combining can improve performance by a factor of 4 or more when used for programs with producer/consumer data on machines with hundreds of processors.

This research was supported under ONR Contract No. N00014-92-J-1801 (in conjunction with the ARPA HPCC program, ARPA Order No. 8930) and NSF CISE Institutional Infrastructure Program Grant No. CDA-8822724. Ricardo Bianchini is supported by Brazilian CAPES and NUTES/UFRJ fellowships.

REPORT DOCUMENTATION PAGE

Form: Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED

4. TITLE AND SUBTITLE

Eager Combining: A Coherency Protocol for Increasing Effective Network and Memory Bandwidth in Shared-Memory Multiprocessors

5. FUNDING NUMBERS

N00014-92-J-1801 //
ARPA HPCC Order 8930

6. AUTHOR(S)

Ricardo Bianchini and Thomas J. LeBlanc

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Computer Science Dept.
734 Computer Studies Bldg.
University of Rochester
Rochester, NY 14627-0226

8. PERFORMING ORGANIZATION REPORT NUMBER

TR 485 .

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Office of Naval Research
Information Systems
Arlington, VA 22217

ARPA
3701 N. Fairfax Drive
Arlington, VA 22203

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Distribution of this document is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

(see title page)

14. SUBJECT TERMS

non-uniform distribution of accesses; memory and network bandwidth; contention; scalable multiprocessors; coherency protocols

15. NUMBER OF PAGES

24

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
unclassified	unclassified	unclassified	UL

1 Introduction

One common cause of poor performance in large-scale shared-memory multiprocessors is limited memory or interconnection network bandwidth. Even well-designed machines can exhaust the available bandwidth when a program issues an excessive number of remote memory accesses or when remote accesses are distributed non-uniformly. While techniques for improving locality of reference are often successful at reducing the number of remote references, a non-uniform distribution of references may still result, which can cause contention both in the interconnection network and at remote memories.

A non-uniform distribution of remote accesses can be caused by a variety of common data sharing patterns. Centralized spin locks, for example, distort the distribution of accesses in a program by forcing all processors to access the memory module containing the lock data structure. In many linear algebra algorithms, including a straightforward parallelization of Gaussian elimination and the LU decomposition algorithm in [Mowry and Gupta, 1991], a single processor writes a row of a matrix which must then be read by every other processor. Classical graph algorithms, such as transitive closure and all-pairs shortest path, exhibit this same structure. Many optimization algorithms, such as LocusRoute [Rose, 1988] and parallel genetic algorithms [Bianchini and Brown, 1993], maintain the best global solution found so far in a global location, and require all processors to access that location each time a new solution is found. These sharing patterns can introduce enormous network and memory contention on large-scale machines, since a large number of processors may need to access a single memory module simultaneously.

These sharing patterns are all specific instances of producer/consumer data [Bennett *et al.*, 1990] (or mostly-read data [Weber and Gupta, 1989]), where data is written by one processor and then read by many processors. Several techniques have been developed for reducing the contention caused by producer/consumer sharing. Hardware combining in the NYU Ultracomputer [Gottlieb *et al.*, 1983] and the IBM RP3 [Pfister *et al.*, 1985] can alleviate contention for spin locks by combining requests to a single memory location. Alternatively, spin locks can be implemented so as to spin on local memory only, thereby eliminating most remote references associated with synchronization [Mellor-Crummey and Scott, 1991]. Optimization algorithms can avoid contention by examining or updating the global solution infrequently. Linear algebra algorithms can exploit the properties of numerical equations to improve locality of reference, and as a side-effect eliminate most producer/consumer sharing [Gallivan *et al.*, 1990].

Although most of these techniques reduce contention and improve locality of reference, they may introduce significant complexity in the algorithm, and do not generalize to all producer/consumer sharing. For example, the block algorithms used in linear algebra are quite complex (compared to the straightforward algorithms), and do not generalize to graph algorithms. Optimization algorithms that avoid using the global solution so as to improve locality of reference may adversely affect the search [Bianchini and Brown, 1993]. Given the frequency with which producer/consumer sharing arises, the performance implications for large-scale machines, and the complexity of eliminating this sharing pattern on a case-by-case basis, a general solution is desirable.

In this paper we examine the effect of producer/consumer data on network and memory contention in large-scale, direct-connect, distributed-shared-memory multiprocessors like the Stanford DASH [Lenoski *et al.*, 1993] and MIT Alewife [Agarwal *et al.*, 1992] machines. We use detailed execution-driven simulation of parallel programs to quantify the performance impact of

producer/consumer data as a function of the network and memory bandwidth and the number of processors in the machine. Our experiments show that, over a wide range of network and memory bandwidths (including the design space of most existing machines), contention caused by producer/consumer data can severely degrade application performance, and the negative effects of contention increase substantially with an increase in processors. Our results also show that, although memory bandwidth is an important resource for this class of programs, the network bandwidth at each processing node is the limiting factor in performance when the network links and memory have comparable bandwidth.

As a solution to the problem of producer/consumer data sharing, we propose a new coherency protocol, called *eager combining*. This protocol replicates a producer's data among multiple memory modules, thereby effectively increasing both the memory and network bandwidth of the producer, and dramatically decreasing the remote access latency of consumers. We discuss the advantages of this technique over other approaches for reducing contention for producer/consumer data, and use simulation to show that eager combining can improve performance (and effective bandwidth) by a factor of 4 or more when used for programs with producer/consumer data on machines with hundreds of processors.

The remainder of this paper is organized as follows. In Section 2 we review related work concerned with referencing patterns and contention in multiprocessors, and place our work in perspective. In Section 3 we describe our multiprocessor simulator and application suite. In Section 4 we quantify the effects of contention as a function of network and memory bandwidth. Sections 5 describes the protocol in detail. In Section 6 we evaluate the performance of our applications under the eager combining protocol, and compare it to software logarithmic broadcasting. We conclude, in Section 7, with a summary of our results.

2 Relationship to Previous Work

A uniform distribution of accesses (and therefore uniform utilization of memory modules) is a common assumption in analytical models used to guide multiprocessor and network design. In [Agarwal, 1992], for example, this assumption is used in the calculation of the round-trip latency of a non-local memory reference, which is then used in the calculation of the number of processor contexts for a multi-threaded multiprocessor. In this paper we show that this assumption is not valid for many programs, and that non-local memory references are considerably more expensive when a non-uniform distribution of references results in contention.

Most previous studies of non-uniform addressing and contention have focused primarily on the effects of hot spots on multistage interconnection networks [Pfister and Norton, 1985; Patel and Harrison, 1988]. These studies focused on eliminating tree saturation, and used synthetic applications for experiments. Our work focuses on direct-connected, distributed-shared-memory machines, and our experiments are based on real application programs.

Glenn *et al.* [Glenn *et al.*, 1991] studied hot-spot effects in synthetic applications in the absence of network congestion and processor caches. They divided hot spots into three categories: 1) a read-only memory location with a large number of readers; 2) synchronized access to memory modules due to related strides; and 3) hot spots caused by synchronization references. This classification does not apply directly to machines with processor caches, which do not exhibit type 1 hot spots, and

may not exhibit type 3 hot spots given an appropriate implementation of synchronization [Mellor-Crummey and Scott, 1991]. In addition, type 2 hot spots are primarily an issue on machines that use low-order interleaving of addresses. In this paper we focus on a different type of hot spot, which is caused by producer/consumer sharing of data.

Previous studies have considered the relationship between a program's sharing patterns and contention. Eggers and Katz [Eggers and Katz, 1988] compared the coherency overhead of write-update and write-invalidate protocols on small-scale multiprocessors, and observed that contention for locks and data was not significant in their applications. Gupta and Weber [Gupta and Weber, 1992] classified data objects according to their expected cache invalidation behavior, and showed that for some objects (i.e., synchronization objects, mostly-read objects, and objects that are frequently read or written) the average number of invalidations per shared write increases with the number of processors. This result suggests that access to these objects may generate contention when the program is run on a large-scale machine. Neither of these studies considered large-scale machines however; our simulations of parallel programs on multiprocessors with hundreds of nodes indicate that the contention associated with producer/consumer data can dramatically increase the running time of a program.

Munin [Carter *et al.*, 1991] is a runtime system for distributed-memory machines that supports alternative software coherence schemes for different types of shared objects, including producer/consumer data. Munin's software coherence protocol for producer/consumer objects is based on object replication and write-update. The Munin implementation runs on SUN workstations on an Ethernet. By way of contrast, our eager combining protocol is an extension of the DASH write-invalidate protocol, and is intended for use in large-scale shared-memory machines with no hardware broadcast mechanism. Our protocol incorporates ideas from software combining trees [Yew *et al.*, 1987] and eager sharing [Wittie and Maples, 1989] to distribute requests for producer/consumer data throughout the machine.

In [Bianchini *et al.*, 1993], we addressed the problem of producer/consumer data at the application (compiler) level through a technique called block-column allocation. Although successful at spreading memory requests and alleviating contention, block-column allocation is only effective for certain types of matrix computations. In this paper we explore a more widely applicable solution that does not depend on sophisticated compilers or programmers.

3 Simulation Methodology and Workload

We are interested in exploring variations in bandwidth and cache coherency protocols in large-scale shared-memory multiprocessors, and therefore direct experimentation is not an available option. Thus, we use simulation for our studies.

3.1 Multiprocessor Simulation

We use an execution-driven on-line simulator that exploits a mixture of interpretation and native execution to simulate unmodified MIPS R3000 object code. The simulator is divided into two parts, an event generator [Veenstra, 1993] and an event executor. The event generator simulates the processor and registers and calls the event executor on every memory reference. The event executor

processes each reference, returning immediately on a cache hit, or passing the reference through the interconnection network to a remote node in the case of a cache miss. The event executor determines which processors should be blocked awaiting remote references and which processors may continue to execute. We simulate events at the level of processor cycles; all simulation parameters and results are expressed in terms of processor cycles.

Our event executor deals with all the major components of a parallel computing system: caches (including TLBs), the interconnection network, local memories, and directories.

We simulate a large-scale (up to 256 processors) direct-connected multiprocessor. Each node in the simulated machine contains a single processor, cache memory, local memory, directory memory, and network interface. Each processor has an infinite, write-back cache with 32-byte blocks. Caches are kept coherent using an implementation of the DASH protocol [Lenoski *et al.*, 1990] with release consistency.

The simulator implements a full-map directory for controlling the state of each block of memory. Each node contains the directory for the memory associated with that node.

Throughout this paper we refer to the ensemble of addressable local memory and directory memory at each node as a "memory module." We simulate three types of memory systems: memory modules that respond with a negative acknowledgement if the memory is busy; memory modules that queue requests (coming either from the cache or network interface) when the memory is busy; and infinitely-ported contention-free memory modules. An infinitely-ported memory module can satisfy an arbitrary number of memory requests simultaneously, but each request is delayed by the memory service time (latency per cache block). We vary the memory latency per block between 8 and 32 cycles. Each shared address space consists of 4KB pages. The pages of an address space are allocated to processors in round-robin fashion.

The interconnection network is a bi-directional wormhole-routed mesh, with dimension-ordered routing. At any node of the machine, the connection between the switch node and the processing node's network interface has the same bandwidth as any network link. Thus, we will refer to the the bandwidth of that connection as the *network bandwidth per node*. The network clock speed is the same as the processor clock speed. Switch nodes introduce a 2-cycle delay to the header of each message. The bandwidth of the network is a parameter in our study. In our finite-bandwidth networks (derived from the Alewife cycle-by-cycle network simulator), contention for network links and buffers is fully captured. The network interface has a queue for out-going messages (triggered either by the cache or the memory module in the node). For comparison purposes, we also implement an idealized, contention-free network, where the link width determines the bandwidth available to an individual packet, but any number of packets can traverse the same link or be stored in the same buffer simultaneously.

Synchronization events do not generate memory or network traffic in our machine model, although they are used to maintain the relative timing of events. We ignore the traffic associated with synchronization so as to avoid having our results dominated by a poor implementation of locks or barriers.

3.2 Workload

Our application workload consists of five parallel programs with producer/consumer data sharing: two linear algebra applications (Gaussian elimination and matrix inversion), two graph algorithms

(transitive closure and all-pairs-shortest-paths), and an optimization algorithm (simulated annealing).

Our implementation of Gaussian elimination is similar in structure to the LU-decomposition application in [Mowry and Gupta, 1991]. The input is a 512 × 512 matrix of linear equations. During each phase of the algorithm, processors need access to a pivot row of the matrix. This pivot row is written by one processor and then read by every other processor. The elements of a row are allocated to consecutive addresses in a single memory module, so all processors direct a request to the same memory module after synchronizing. Synchronization is implemented with locks.

Our implementation of matrix inversion uses an input matrix of size 512 × 512. The first phase of matrix inversion uses LU-decomposition, which has roughly the same structure as Gaussian elimination. Although the second phase requires that all processors access the same row of the matrix, most of these accesses will result in local cache hits. We use barriers for synchronization.

In our implementation of transitive closure, each process operates on a set of rows of an adjacency matrix stored in row-major order. Our input graph has 768 vertices, and each vertex is connected to each other vertex with probability 0.5. As with Gaussian elimination, the algorithm operates in phases, where processors need access to the same row of the adjacency matrix at the start of a phase. Unlike Gaussian elimination, not every processor needs access to the row; the input determines how many processors must access the pivot row in a phase.

We use a straightforward parallelization of Warshall-Floyd's algorithm to compute the all-pairs shortest paths of a graph. The input graph has 512 vertices, and each vertex is connected to each other vertex with probability 0.5. Although this application is similar in structure to transitive closure, the two programs differ in the amount of communication required and the style of synchronization. All-pairs shortest paths represents distances using 4-byte integers, while transitive closure uses single bytes to represent connectivity. Thus, there is more communication required in all-pairs shortest paths, even though both programs do roughly the same amount of work. In addition, we use barriers for synchronization in all-pairs shortest paths, and locks for synchronization in transitive closure. Barriers are much more prone to contention, since they cause processors to access producer/consumer data in lock-step.

Our simulated annealing program finds the minimum of a complex function of 8 variables. In this program, processors search the solution space independently, periodically examining the current best-known solution, which is stored in a global location. The probability that a processor examines the global solution increases with time; processors rarely read the global solution in the beginning of the computation, and rarely write the solution near the end of the computation.

4 Effect of Network and Memory Bandwidth on Contention

In this section we quantify the effects of contention that result from producer/consumer sharing as a function of network and memory bandwidth. We show that available bandwidth is the limiting resource for programs with this sharing pattern, and that simply increasing absolute bandwidth is not likely to provide a cost-effective solution to the problem.

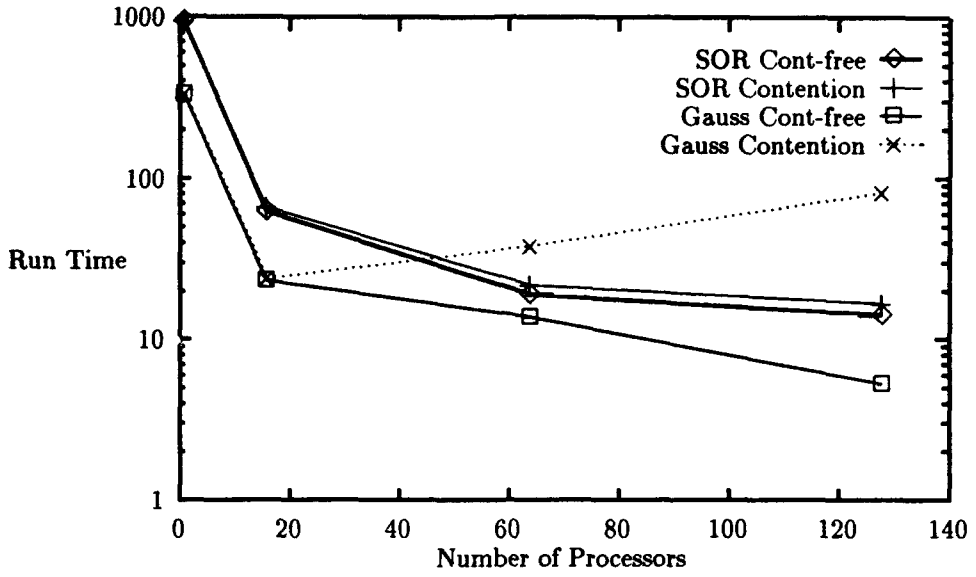


Figure 1: Running time (M cycles) of SOR and Gaussian Elimination.

4.1 Effect of Non-Uniform Distribution of References

Figure 1 compares the running time of SOR and Gaussian elimination on a multiprocessor with low memory and network bandwidth (200 MB/second, assuming a 100 MHz clock) to the running time on an ideal contention-free multiprocessor. SOR implements successive over-relaxation of a 768 $\times$ 768 matrix. Blocks of rows of the matrix are assigned to each processor. Processors work roughly independently of each other; communication only takes place when processors update the elements of their boundary rows. The application is executed for 50 iterations, which are divided into pairs of phases separated by barriers.

From the figure, we observe that the running time of SOR is not affected by contention; the machine with limited memory and network bandwidth performs comparably to the contention-free machine. The running time of Gaussian elimination is affected by contention however; the program can utilize fewer than 20 processors on the limited-bandwidth machine, but can exploit over 120 processors on the contention-free machine.

Both SOR and Gaussian elimination have very low miss rates – around 2% on 128 processors. Thus, both programs have good locality of reference. In general, we would expect programs with very low miss rates to perform well on large-scale machines (assuming sufficient parallelism and appropriate synchronization). The problem with Gaussian elimination is not the number of remote references, it is the non-uniform distribution of those references. Programs with good locality and nearest-neighbor communication, like SOR, do not saturate the network or memories. Programs with good locality and a highly non-uniform distribution of references can saturate both the network and memories.

This experiment shows that it may be difficult to design a multiprocessor that can run all

Machine	Path Width	Latency/Switch	Latency/Link	Bi-dir Link Bandwidth
Contention-free	32/16/8 bits	2 cycles	1 cycle	Infinite
High	32 bits	2 cycles	1 cycle	800 MB/sec
Medium	16 bits	2 cycles	1 cycle	400 MB/sec
Low	8 bits	2 cycles	1 cycle	200 MB/sec

Table 1: Network Bandwidth

Machine	Latency/Word	Queue Size	Memory Bandwidth
Contention-free	1/2/4 cycles	0 entries	Infinite
High	1 cycle	16 entries	400 MB/sec
Medium	2 cycles	16 entries	200 MB/sec
Low	4 cycles	16 entries	100 MB/sec

Table 2: Memory Bandwidth

classes of applications efficiently. Even well-designed machines, with seemingly sufficient network and memory bandwidth, may perform poorly for a large class of applications due to bandwidth limitations. To quantify the relationship between available bandwidth and contention, we will explore the parameter space of network and memory bandwidths, and investigate how application performance varies with bandwidth.

4.2 Isolating Memory and Network Contention

We consider three levels of finite bandwidth for the network and memories (cache bandwidth is assumed higher than both memory and network bandwidth). The bandwidth and latency parameters used in our experiments are described in Table 1 and 2. We simulate machines that represent a range of "balanced" architectures, where the memory bandwidth is equal to the uni-directional bandwidth of a network link. (We assume 100 MHz clocks in all machines.) Our low-bandwidth machine has memory and (uni-directional) network bandwidth of 100 MB/second; the medium-bandwidth machine has twice as much bandwidth as the low-bandwidth machine; the high-bandwidth machine has twice as much bandwidth as the medium-bandwidth machine. Our contention-free machine simulates a contention-free network and contention-free memories.

Our first set of results isolates the contribution of limited memory bandwidth to performance degradation, when the network bandwidth per node is relatively high. In these experiments, we simulated Gaussian elimination on a machine with a contention-free network. Figure 2 plots the running time of Gaussian elimination on 64 and 128 processors as a function of memory bandwidth. [1] We can see from this figure that, in the absence of network and memory contention, the running time is not significantly affected by changes in the memory latency per cache block; the latency of the memory is only a small part of the latency of a remote reference, which is dominated by the

[1] In these simulations we increase the memory bandwidth by decreasing memory latency.

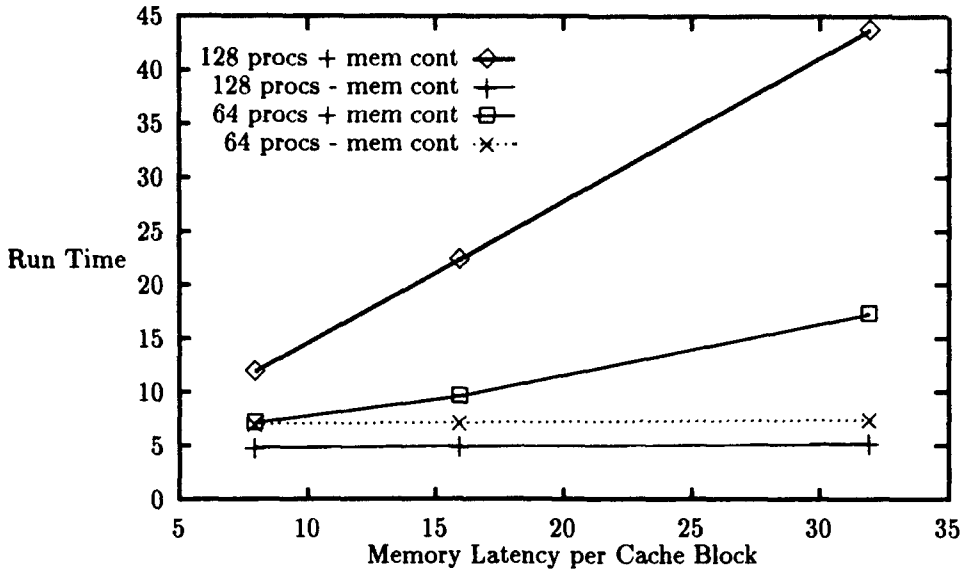


Figure 2: Running time (M cycles) of Gaussian elimination - contention-free network

round-trip time through the network. However, when we include the effects of memory contention, we see that increasing memory latency per cache block (and therefore effectively decreasing memory bandwidth) has an enormous effect on the running time, especially on 128 processors.

It is interesting to observe that bandwidth limitations change the relative performance of 64 and 128 processor configurations. That is, a 128-processor machine executes the program faster than a 64-processor machine in the absence of contention. We observe speedup when moving to the larger machine in the contention-free case, but the same move causes a massive slowdown when bandwidth limitations are considered.

In our next set of results, we assume contention-free memories, so as to isolate the contribution of network congestion to performance degradation when there is significantly more memory bandwidth than network bandwidth. In these experiments, we simulated Gaussian elimination on a machine with contention-free memories; the results appear in figure 3. The large gaps between the curves for contention-free machines and the corresponding curves for bandwidth-limited machines indicates that, in the absence of memory contention, network bandwidth represents a serious bottleneck. Once again, limited bandwidth changes the relative performance of the 64-node and 128-node machines in comparison to the contention-free case.

Sophisticated routing techniques, such as randomized and adaptive routing, are of no help in reducing the network contention we observe here. Most of the overhead attributed to network contention in figure 3 is caused by two factors: (1) long messages (i.e., read replies) leaving the producer's memory module contend for the single link connecting the producer's network interface to the network switch node, and (2) heavy network traffic around the producer's node results in contention for nearby network links. Sophisticated routing schemes cannot resolve the intrinsic problem that the path between a single memory module and the network is saturated.

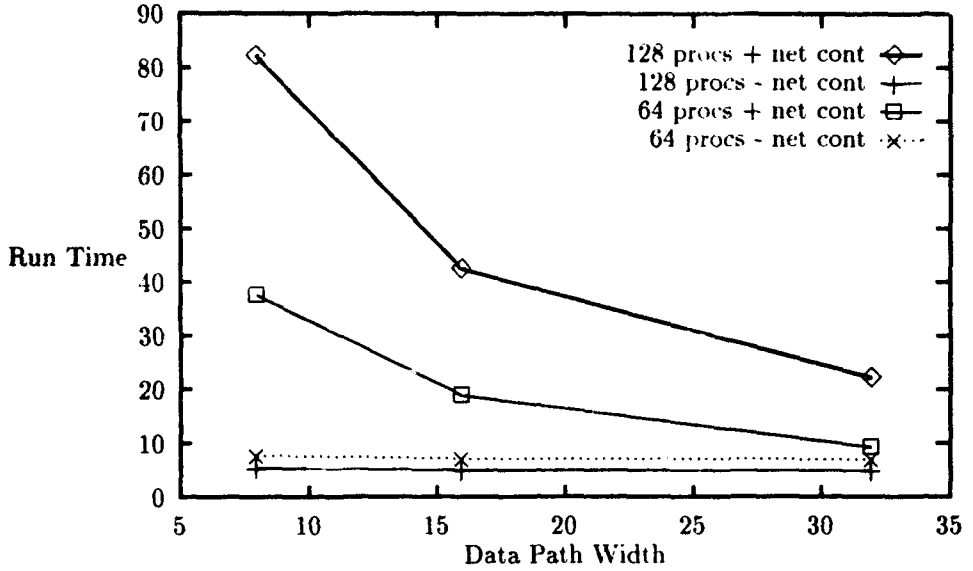


Figure 3: Running Time (M cycles) of Gaussian Elimination - contention-free memories

Network	Memory Bwidth											
Bwidth	High				Medium				Low			
	RT	CFRT	RL	CFRL	RT	CFRT	RL	CFRL	RT	CFRT	RL	CFRL
High	22.4	22.3	55.3	50.2	22.5	22.4	62.9	56.1	23.1	22.7	89.6	68.0
Medium	22.7	22.5	70.8	59.4	22.8	22.6	76.1	65.3	23.3	22.9	99.4	77.0
Low	23.7	22.9	119.3	80.2	23.7	23.0	125.0	85.9	23.9	23.3	137.3	97.7

Table 3: Running Times and Avg. Latencies with and without Contention of Gauss on 16 procs

4.3 Scaling the Number of Processors

In tables 3-5 we examine the aggregate effect of the two types of contention on the performance of Gaussian elimination using more realistic scenarios of limited network and memory bandwidth. These tables present the running time of the program on realistic machines with finite network and memory bandwidth (RT), the running time in the absence of network and memory contention (CFRT), the average remote access latency on the realistic machines (RL), and the average remote access latency on the contention-free machines (CFRL). Running times are given in millions of cycles.

It is interesting to observe the evolution of the contention problem as we increase the size of the machine from 16 to 64 to 128 processors. On 16 processors, contention for the network and memories is virtually nonexistent. In each case the running time under limited bandwidth is roughly the same as the running time on a contention-free machine. In addition, the difference in running time between the machine with high memory and network bandwidth and the machine with low memory and network bandwidth is only about 7%. Although the average remote access latency on these machines differs by a factor of 2-3, the miss rate is small enough that this factor does not significantly affect running time.

Network	Memory Bwidth											
Bwidth	High				Medium				Low			
	RT	CFRT	RL	CFRL	RT	CFRT	RL	CFRL	RT	CFRT	RL	CFRL
High	9.0	6.9	181.5	66.6	13.6	7.1	438.2	74.1	26.4	7.3	1126.1	89.2
Medium	19.1	7.1	767.2	76.6	19.0	7.4	758.4	85.0	27.3	7.5	1185.0	99.4
Low	37.5	7.6	1812.5	99.1	37.6	7.7	1814.6	107.9	37.9	8.0	1828.2	122.5

Table 4: Running Times and Avg. Latencies with and without Contention of Gauss on 64 procs

Network	Memory Bwidth											
Bwidth	High				Medium				Low			
	RT	CFRT	RL	CFRL	RT	CFRT	RL	CFRL	RT	CFRT	RL	CFRL
High	23.8	4.8	1201	81.3	29.4	4.9	1507	88.9	54.1	5.2	2847	103.9
Medium	43.2	5.0	2366	91.7	43.4	5.1	2368	99.1	60.4	5.4	3233	114.1
Low	82.4	5.3	4656	114.9	81.3	5.5	4651	122.3	82.8	5.7	4676	137.4

Table 5: Running Times and Avg. Latencies with and without Contention of Gauss on 128 procs

On 64 processors, the running time on the limited bandwidth machines is substantially higher than the running time on the corresponding contention-free machines, by 30% on the machine with the highest memory and network bandwidth and 470% on the machine with the lowest memory and network bandwidth. Whereas a factor of four increase in processors (from 16 to 64 processors) produces a factor of 2.5 improvement in running time on the machine with high memory and network bandwidth, the same increase in processors results in a higher running time on the machines with low memory or network bandwidth, and no significant increase in running time on the machines with medium network bandwidth. The effect of contention is illustrated most dramatically by the average remote access latency, which increases from 123 cycles to over 1800 cycles on the machine with low memory and network bandwidth.

These trends continue as we increase the number of processors from 64 to 128. In the absence of contention, a factor of two increase in processors produces a factor of 1.4 improvement in running time on most of the machines. When contention is included, the running time increases by a factor of 3 or more in nearly every case. Even on the high bandwidth machines, the remote access latency is well over 1200 cycles.

We can make a general observation from these tables: the network is the bottleneck in machines in which the uni-directional network bandwidth is equal to or less than the memory bandwidth, while the memories are the bottleneck in the remaining cases. As we reduce memory bandwidth and keep the network bandwidth constant (that is, as we move across a row of the tables), running time remains roughly constant until the memory bandwidth becomes strictly less than the network bandwidth. At that point, the running time increases, since it is dominated by memory contention effects. If we increase the network bandwidth while keeping the memory bandwidth fixed (that is, as we move up a column in the tables), the running time improves greatly whenever memory bandwidth is greater or equal to network bandwidth.

These results suggest that the usual notion of balanced architecture, in which memory and (unidirectional) network bandwidth are roughly equivalent, exhibits a serious bottleneck at the producer's network link. There are two reasons for this. First, in addition to the data produced by the memory or cache, the network must also transfer other information, such as headers, addresses, and operation codes. Second, both the memory and cache controller compete for access to the network at each node of the multiprocessor.

It is apparent from these results that producer/consumer data sharing can introduce enormous contention on large-scale machines. The problem is that only an inordinate amount of bandwidth can eliminate the contention levels observed here. Rather than build machines with sufficient bandwidth to handle extreme cases of non-uniform distributions of references, we should consider ways to provide higher *effective* bandwidth.

4.4 Dealing with Limited Bandwidth

There are several alternatives for dealing with insufficient bandwidth caused by producer/consumer sharing of data. The programmer can modify the application so as to reduce the frequency of remote references, avoid producer/consumer data sharing, or efficiently copy the producer's data throughout the machine. Alternatively, hardware designers can provide more absolute bandwidth, longer queues at the network and memory modules, or efficient hardware replication strategies.

Reducing the frequency of remote references in the application is typically a complex and ad hoc task, which may or may not result in a uniform distribution of references. As seen in the previous section, even programs with a very low miss rate can suffer serious performance degradation due to bandwidth limitations. A new algorithm may be required to avoid producer/consumer sharing, and this could affect the application's results (as in the case of optimization algorithms that avoid examining the global solution because of contention).

Increasing the queue size for busy memory modules reduces the need for negative acknowledgements, thereby lowering the bandwidth demands of the application. In addition, longer queues allow a memory module to be continuously utilized during periods of contention. However, our simulation results show that the memory module is already fully utilized during periods of contention when the queue holds 16 entries; longer queues would not help increase memory utilization. [2] Furthermore, negative acknowledgements have little effect on performance, unless network bandwidth is extremely limited, or the queue size is so small that most requests generate a negative acknowledgement.

Increasing the bandwidth of the machine is not a feasible option. Our simulations show that, even at very high levels of memory and network bandwidth, it is not possible to keep up with the bandwidth requirements of our applications on large-scale machines.

There are two ways to increase the *effective bandwidth* of the machine: memory interleaving and data replication. Interleaving shared addresses, as in the IBM RP3 [Pfister *et al.*, 1985], improves performance by reducing the number of memory conflicts for shared data. In [Bianchini *et al.*, 1993], we showed that interleaving of cache blocks across memories can significantly improve performance in our producer/consumer programs, even when implemented at the software (compiler) level.

[2] This reasoning also applies to back-off strategies, in which a processor rejected by a busy memory module waits for a small random period of time before reissuing the request. Random back-off is likely to leave memories underutilized during periods of contention, and, at best, will approach the performance of memory queues.

Data replication can be either producer-driven or consumer-driven. We refer to producer-driven approaches as *eager sharing*. In eager sharing, the producer of a data block copies it to the nodes that need it. An efficient implementation of eager sharing requires that we 1) identify the recipients of the data to be multicast and 2) multicast the data block only when the producer is finished modifying the entire block. Examples of eager sharing approaches include write-update protocols, post-store (producer prefetch) instructions, delayed update protocols, and software logarithmic broadcasting.

Write-update protocols [Archibald and Baer, 1986] can be used to provide consumers with data before they request it, thus reducing the need for requests, and the memory and network bandwidth they consume. However, a write-update protocol in a machine with no broadcast medium would likely overload the home node when many processors share the data. In addition, write-update protocols typically operate at the level of individual words, generating a network message on *every* write to shared data. In many cases, this extra message traffic can become a serious problem, outweighing the benefits of this type of protocol. For this reason, write-update protocols are most appropriate for small data items [Lenoski *et al.*, 1993].

Post-store instructions [Rosti *et al.*, 1993] are similar to write-update in that they update all cached copies of the data. The use of post-store is controlled in software however, and post-store instructions apply to entire cache blocks. Although post-store instructions generate less traffic than a write-update protocol they would still overload the producer's node in the absence of a broadcast medium.

Operating system approaches to data replication, such as the delayed updates of Munin [Carter *et al.*, 1991], are more flexible than the hardware schemes mentioned above, but must incur the overhead of handling sharing in software. In addition, Munin (like most other systems that support write update) is implemented on a broadcast medium (i.e., Ethernet).

Replication can also be implemented at the application or runtime level by having the producer broadcast the data to all consumers. There are two problems with this approach: it requires a distributed-memory programming style in which coherency of shared data is not guaranteed by the hardware, and the pattern of sharing between processors must be easily predictable. When coherency is not required and applications are well behaved, software logarithmic broadcasting performs well.

Consumer-driven data replication approaches are referred to as *combining trees* [Yew *et al.*, 1987]. In this approach, processors are organized into a tree structure, in which each processor requests the data block from its parent node, with the producer at the root. Each parent node combines the requests of its children into a single request to its parent.

The relative feasibility and performance of the producer and consumer-driven styles of data replication is dependent on the communication and addressing features of the underlying architecture. Scalable distributed-shared-memory architectures can be separated into two categories, based on whether a data block has a home. Architectures in which data has no fixed home are referred to as COMA (Cache Only Memory Architecture), e.g., the KSR1 [Kendall Square Research Corp., 1992]. Architectures in which data blocks have a home are referred to as CC-NUMA (Cache-Coherent Non-Uniform Memory Access), e.g., DASH [Lenoski *et al.*, 1990].

COMA machines are usually based on broadcasting media, since processors must be able to easily locate any data blocks they access. For example, the KSR1 uses a hierarchy of rings; a

processor broadcasts requests for data blocks on the local ring, which are forwarded up the hierarchy of rings if necessary to satisfy the request. Once data is brought into a local ring, subsequent requests for the data are satisfied by the local ring. In effect, the KSR1 implements a dynamic combining tree solution to memory contention, limiting it to at most 32 processors (the number of processors on the local ring). Similar combining capabilities are provided by hierarchical cache/bus architectures, such as presented in [Wilson Jr., 1987].

CC-NUMA machines, on the other hand, usually have no such broadcast medium, making it more difficult to either locate replicated data (in a combining tree approach) or to perform dynamic multicast (in an eager sharing approach). In the following section we describe a modification to the DASH coherency protocol that implements data replication in a CC-NUMA machine. We show how this protocol solves the problems associated with the lack of a broadcast medium, and then show that the protocol greatly increases effective memory bandwidth and reduces the need for network bandwidth in producer/consumer programs.

5 Eager Combining Coherency Protocol

In this section we describe a coherency protocol that implements data replication for hot spots caused by widespread producer/consumer sharing. Our goal is to increase effective memory bandwidth and decrease the need for network bandwidth in direct-connected, distributed-shared-memory multiprocessors.

We assume certain physical address ranges are marked *hot*, and these addresses are treated specially by the coherence protocol. Two ways to accomplish this are:

- The choice of protocol could be selected on a per-page basis; an analogous feature is already present in the MIPS R4000 cache coherence controller [MIPS Computer Systems Inc., 1991].
- A portion of the physical address space of the machine could be permanently set aside for hot data.

Which of these approaches is chosen depends on tradeoffs involving cache memory cost, the benefits of dynamic protocol selection, and the ability of the compiler or programmer to precisely identify hot data ranges. Our simulation results assume that all shared data is marked as *hot*.

Our basic approach is to designate a fixed number of "server nodes" for each hot physical page, assigning to each server some subset of the remaining nodes as clients. The protocol uses eager sharing to distribute data to servers, which then satisfy requests from multiple client nodes. Multiple requests that cannot be satisfied immediately by a server are combined to reduce the traffic directed to a hot spot. The protocol assumes the release consistency model. Since our approach incorporates the properties of both eager sharing and combining trees, we call it *eager combining*.

We use the DASH cache coherence protocol [Lenoski *et al.*, 1990] as a starting point for our eager combining protocol. Each data block in DASH is assigned to a memory module, and that module's node is referred to as the data block's *home node*. In the eager combining version of the protocol, we designate a fixed number of *server nodes* for each hot data block, which are determined statically from the physical page number. As with regular data blocks, hot data blocks can be in one of three states: *uncached*, *read-shared*, and *modified*. We make three modifications to the DASH protocol in order to handle hot data blocks:

- Reads to a hot data block are directed to a server rather than to the home node;
- When a hot data block makes a transition from *modified* to *read-shared*, the block's home node multicasts the data to the block's servers;
- When a hot data block makes a transition from *read-shared* to *modified*, the clients and the servers must have their copies of the block invalidated.

The following sections describe each type of transaction in detail. Since our experiments assume infinite processor caches, the following description omits the details of how to handle cache replacements.

5.1 Read Requests

When a node makes a read request to a hot data block, the request goes directly to the proper server, which is selected based on the requester's node number and the physical page number. If the server has an unmodified copy of the block, it stores information to the effect that the requesting processor is a new client and sends the block to the requester. If the server does not have a copy of the block, or if the server has modified its copy of the block, the server marks the requester as a client and forwards the client's request to the home node. Subsequent requests from other clients for the same data block are queued at the server until it receives the block from the home node.

Upon receiving a forwarded read request, the home node proceeds according to the state of the data block. If the block is in the *read-shared* state, the home node sends the block to the client directly. If the block is in the *modified* state, the home node forwards the request to the current owner of the block. As in the DASH protocol, the owner transmits the data block to both the requester and the home node. On receiving the updated contents of the block, the home node sends a copy to each of the servers for the block, and sets the state of the block to *read-shared*. The multicast from the home node to the servers can be overlapped with computation on all nodes.

It is important to note that the home node does not multicast a data block to its servers each time the block is written. The multicast takes place only on the transition from *modified* to *read-shared*. Thus, we avoid eager sharing of partially modified data blocks. Nonetheless, eager combining could exacerbate any adverse performance effects caused by fine-grain sharing and false sharing.

A multicast and the corresponding transition to the *read-shared* state are also performed by the home node upon receiving a forwarded read request for an *uncached* hot data block.

5.2 Write Requests

When a client issues a write to a hot data block, a request for ownership (and in some cases, data) is sent directly to the home node, bypassing the server. On receiving this request, the home node proceeds according to the current state of the data block. If the block is in the *modified* state, the protocol proceeds almost exactly as in DASH: the request is forwarded to the current owner of the block, which transfers ownership (and perhaps data) to the requesting node, and requests that the home node update the ownership of the block. Although this latter request does not generate an acknowledgement in the DASH protocol, the eager combining protocol does require an

acknowledgement from the home node to the previous owner of the block, so as to avoid sending ownership update messages through servers to the home node. [3]

We considered having both read and write requests go through servers, which would increase the number of messages for write requests, but would result in a single path between clients and the home node. We send write requests directly to the home node in order to reduce the traffic directed to server nodes, and to avoid increasing the number of messages involved in requests whenever possible.

If the write finds the data block in the *read-shared* state, the home node must invalidate all copies, both in the servers and clients. To implement these invalidations, the home node sends invalidation messages to servers, who then pass on invalidations to their clients. In this scheme, the directory information in the home node is consistent with respect to the state of a data block and the number of servers, but not the number of clients. When a write request reaches the home node, the home sends the data to the new owner, and tells the new owner the number of servers caching copies of the data block. The home node sends invalidation messages to each sever, which then send invalidation messages to each client. When the clients have all acknowledged the invalidation to their server, the server sends an acknowledgement to the new owner. The home node invalidates the previous owner of the data block, which then acknowledges the new owner directly. [4]

In comparison to the DASH protocol, this invalidation scheme reduces the total amount of work required of the home node to service hot data blocks (under the assumption that servers for a data block usually access the data block). Instead of distributing copies of a hot data block to all processors (as in DASH), the home node need only send copies to the servers for the block. Most read requests are satisfied by servers, and therefore never reach the home node. If a server must forward a read request to the home node on behalf of one client, that request will generate a single response that will satisfy requests from the other clients of the server. In general, this scheme reduces the number of messages the home node must send, since the number of servers for a hot block is expected to be much smaller than the number of processors using the block.

5.3 Protocol Overhead

Eager combining is not without costs. It introduces additional messages and requires extra space in the directory. We will now consider how much overhead is associated with the protocol.

Tables 6 and 7 present a comparison between the number of messages involved in the DASH protocol and in eager combining. In these tables, S stands for the number of servers per hot data block, C is the total number of clients of the hot block, and SC is the number of servers that are also clients of the hot block. The number of hops referred to in the tables is the number of

[3]The reason for this extra acknowledgement message is that the ownership update message and a subsequent read request by the same processor may arrive out-of-order at the home node, due to the different paths these messages take through the network. (The read request would go to a server first.) Requiring the previous owner to wait for an acknowledgement before issuing any other read request to the same block guarantees correct message ordering. This message does not have a noticeable impact on performance, since the extra acknowledgement is not in the critical path.

[4]The home node must invalidate the previous owner since the identity of this owner is unknown to the servers, since this processor keeps a clean copy of the data on *modified* to *read-shared* transitions, instead of having to access a server for such a copy.

Consistency Model	Protocol	Total Number of Messages	Number of Hops
Sequential	EC	(2 * S + 2 * (C - SC)) + 2	(2 * S + 2 * (C - SC)) + 2
	DASH-like	(2 * C) + 2	(2 * C) + 2
Release	EC	(2 * S + 2 * (C - SC)) + 2	2
	DASH	(2 * C) + 2	2

Table 6: Messages transferred in coherency actions (read-shared to modified)

Protocol	Total Number of Messages	Number of Hops
EC	S + 5	4
DASH	4	3

Table 7: Messages transferred in sharing actions (modified to read-shared)

messages in the critical path of the protocol. As observed in the tables, eager combining may employ more messages than the DASH protocol when making a transition from *read-shared* to *modified* or *modified* to *read-shared*, because hot data blocks are sent to servers whether or not the servers use the data, and both the servers and clients must be kept consistent. These extra messages are unlikely to be a serious problem however, since we expect hot data blocks to be accessed by most processors (including the servers), and the extra messages required to replicate data to servers can be overlapped under any consistency model. Also, it is not necessary to wait for invalidation acknowledgements if sequential consistency is not required. Therefore, under a relaxed consistency model, eager combining does not impose significantly greater communication latency than the DASH protocol. As shown by the number of hops in the critical path in tables 6 and 7, eager combining and DASH require roughly the same number of hops for each state transition, under the assumption that each server actually requires the data it provides to its clients.

It may seem that replicating cache lines in a large number of servers consumes an excessive amount of cache space, but this is not so. Assuming an even distribution of servers throughout the machine, the maximum additional cache space needed per processor is *HotDataSize * NumServers / NumProcs*, where *HotDataSize* is the size of the hot data, *NumServers* is the number of servers per hot cache block, and *NumProcs* is the number of processors in the machine. This is a worst-case analysis however; in practice, a server need only store the hot data currently being referenced by clients. In addition, under the assumption that servers are also clients for hot data, then each server needs a copy of the data anyway. Furthermore, the extra cache space devoted to copies of hot data in servers is a small percentage of the cache space devoted to caching application data. In short, the space overhead of servers is not an impediment to data replication as used in eager combining.

Eager combining also requires additional directory space for servers to keep track of clients. This extra space amounts to a vector of *NumProcs / NumServers* bits per cache block, in which each bit represents a client sharing the block. We believe that this overhead is justified by the performance advantages of the protocol, which will be demonstrated in the next section.

Application	Bandwidth								
	High			Medium			Low		
	RT	ECRT	CFRT	RT	ECRT	CFRT	RT	ECRT	CFRT
Gauss	9.0	7.6	6.9	19.0	8.9	7.4	37.9	13.3	8.0
Matinv	65.8	62.4	58.1	76.1	67.4	59.7	99.9	80.5	63.4
Tclosure	47.7	43.4	43.2	55.4	44.1	43.7	74.0	47.5	44.6
All-pairs	43.6	29.1	25.5	62.3	34.3	27.1	102.0	48.5	30.2
SA	12.3	8.9	9.6	16.3	9.2	9.8	21.5	9.7	10.3

Table 8: Application Performance on 64 processors

Application	Bandwidths								
	High			Medium			Low		
	RT	ECRT	CFRT	RT	ECRT	CFRT	RT	ECRT	CFRT
Gauss	23.8	7.1	4.8	43.4	10.7	5.1	82.8	19.0	5.7
Matinv	64.9	62.9	50.3	86.9	76.2	53.6	138.5	109.5	61.1
Tclosure	43.2	23.8	23.6	64.6	26.0	24.0	115.2	34.3	24.9
All-pairs	66.0	47.1	34.4	103.4	60.4	37.7	182.2	91.7	45.1
SA	10.0	4.8	5.4	12.4	5.1	5.5	14.4	5.9	5.8

Table 9: Application Performance on 128 processors

6 Performance Evaluation

In this section we evaluate the eager combining protocol. In our simulations of eager combining, requests are rejected (and must be reissued) when the producer's memory module is busy.

We first present performance results comparing eager combining to the limited bandwidth and contention-free machines. We then compare eager combining to software logarithmic broadcasting.

6.1 Application Performance Comparison

In this section, we study the performance of our example programs under eager combining. We consider balanced architectures, where the memory and (uni-directional) network bandwidth are equivalent. Tables 8 and 9 present the running time of the applications (in millions of cycles) on limited bandwidth machines (RT), eager combining (ECRT), and the idealized, contention-free machine (CFRT) on 64 and 128 processors, respectively. We do not include the results for 16 processors, since our earlier results showed that contention for producer/consumer data does not significantly affect performance on small machines.

On the 64-processor machine we use 4 servers per cache block; we use 8 servers per cache block on the 128-processor machine. All the simulations assume release consistency.

Table 8 shows that eager combining significantly improves the performance of all the applications on the low-bandwidth machine with 64 processors. The running time of Gaussian elimination, all-pairs, and simulated annealing is improved by a factor of 2-3. Transitive closure and matrix inversion also exhibit improved performance, although the gains are not as substantial. Transitive closure has less contention than the other applications, since only half the processors (on average) require access to a producer's data. Matrix inversion suffers contention only during the LU-decomposition phase of the algorithm, so any improvements in performance are limited to that phase.

Substantial improvements are also possible on the medium-bandwidth machine for these applications. The improvements are not as great on the high-bandwidth machine, but the running time under eager combining is close to that of the contention-free machine in most cases. In particular, transitive closure performs close to the optimal under eager combining on high- and medium-bandwidth machines, and is within 7% of optimal on the low-bandwidth machine.

Simulated annealing exhibits an anomaly: the running time under eager combining is better than the running time on the idealized, contention-free machine. Under eager combining, writes (including invalidations) take longer to perform. Thus, processors incur many fewer misses on accesses to the global solution under eager combining. Since the program executes for a fixed number of iterations, the change in the number of misses produces an improvement in running time, even though it may adversely affect the search.

The performance improvement under eager combining is even greater on 128 processors, as shown in table 9. Eager combining improves the running time of Gaussian elimination by a factor of 4 on all the machines. Other applications are improved by a factor of 2-3 in most cases. Once again, the running time of transitive closure under eager combining is very close to the optimal running time on an idealized, contention-free machine.

The performance improvements under eager combining are due to an increase in the *effective* network and memory bandwidth of the machine. Without eager combining, long queues can develop both at the network interface and at the memory. These queues result in an enormous increase in remote access latency. For example, consider **all-pairs** on a 128-processor machine with low network and memory bandwidth. This program represents an extreme case, since it uses barriers, which cause processors to execute in lock-step, thereby increasing contention. Without eager combining, the average queue size at the network interface is 44, for an average delay of 1773 cycles. Under eager combining, the average network interface queue size is reduced to 3.4, for a total delay of 115 cycles. The running time under eager combining on a low-bandwidth machine (91.7M cycles) is less than the running time without eager combining on the medium-bandwidth machine (103.4M cycles). The other applications based on locks (Gaussian elimination, transitive closure, and simulated annealing) all run faster under eager combining and low bandwidth than on a machine with high bandwidth and no eager combining. In effect, eager combining multiplies the bandwidth of the multiprocessor by a factor of 2-4 by distributing requests more uniformly in the machine.

Some observations are common to the two tables. For example, the relative performance of eager combining improves in comparison to the limited bandwidth configurations as we decrease bandwidth. However, decreasing bandwidth also causes the performance of eager combining to worsen relative to the optimal running time on an idealized, contention-free machine. This trend simply indicates that the protocol has intrinsic coherency maintenance costs, and that bandwidth is a scarce resource even under eager combining.

By comparing the results in the two tables, we see that eager combining does not always improve performance as we increase the number of processors. Even when speedup can be achieved in the absence of contention, as in the case of Gaussian elimination, eager combining may require more bandwidth than is available. On low-bandwidth machines with a large number of processors, both the producer's node and the servers may lack sufficient bandwidth to satisfy all requests. Adding more servers relieves the problem at the server nodes, but creates the need for more network bandwidth at the home node. These results suggest that eager combining on large-scale machines may require multi-level trees in order to reduce contention at the servers and home node, although doing so would increase the number of hops for references by clients.

6.2 Comparison with Software Broadcasting

In this section we compare the performance of software broadcasting against eager combining. As mentioned earlier, software broadcasting usually performs well, but coherency is not guaranteed by the hardware, and the pattern of sharing between processors must be easily predictable.

We implemented consumer-driven software broadcasting, wherein the producer sets a flag indicating when data is ready, and the consumers copy the data. In this implementation multiple consumers can overlap time spent in the network, so many copy operations can proceed in parallel. Each tree node contains several broadcast buffers, which allows a parent node to continue producing data before its children consume the data.

Our software broadcasting tree implements the same degree of fan-out at all levels in the tree, since the traffic generated at each level is the same. This is not true of eager combining, where the root node in the tree (the home node) must broadcast the data to the servers **and** satisfy any requests from clients that were forwarded by servers before they received the data. For this reason, the number of servers per block under eager combining is chosen so that each server has more clients than the home node has servers. Thus, for software broadcasting on a 64-processor machine, we use a two-level tree with a fan-out of 8; under eager combining we use 4 servers, each with 16 clients. For software broadcasting on a 128-processor machine, we use a two-level tree with a fan-out of 12; under eager combining we use 8 servers, each with 16 clients.

We compared the running time of Gaussian elimination under eager combining and software broadcasting, while varying the network and memory bandwidth of the machine. We simulated a machine with high network and memory bandwidth, and a machine with low network and memory bandwidth, since these machines represent the best and worst case for eager combining. Under software broadcasting we assume each memory module has a 16-entry queue; under eager combining, requests are rejected when the memory module is busy. The results of these experiments are presented in table 10.

The table presents the running time of the application (in millions of cycles) on a limited-bandwidth machine (RT), under eager combining (ECRT), under software broadcasting (BRT), and on an idealized, contention-free machine (CFRT). As seen in the table, eager combining performs better than software broadcasting in these experiments. There are several reasons for this. Software broadcasting requires more computation, since processors must explicitly perform data copying. In addition, there is more synchronization overhead under software broadcasting, which must synchronize access to the buffers. Most importantly, in a broadcasting tree, a processor at the bottom of the tree cannot trigger a copy operation higher up in the tree; it must wait for its

Bandwidth	Num Procs	RT	ECRT	BRT	CFRT
High	64	9.0	7.6	10.4	6.9
	128	23.8	7.1	10.0	4.8
Low	64	37.9	13.3	18.1	8.0
	128	82.8	19.0	21.8	5.7

Table 10: Broadcasting vs Eager Combining for Gaussian Elimination

ancestor to copy the data, before it can copy the data. Under eager combining, a read operation by any client causes the data to be multicast to the servers, and also forwarded to the client; no processor is forced to wait for an ancestor in the tree.

7 Conclusion

In this paper, we examined the performance implications of a non-uniform distribution of memory accesses caused by producer/consumer sharing of data. Using execution-driven simulation, we observed the effect of variations in memory and network bandwidth on performance. At all levels of bandwidth we considered, our application suite exhibited massive performance degradation on large-scale multiprocessors. Both memory and link bandwidth are limiting factors for our applications. We observed that when memory bandwidth and the unidirectional network bandwidth are comparable (as in most "balanced architectures"), a serious bottleneck may develop at the network interface of the nodes.

To address these bandwidth problems, we proposed the eager combining coherency protocol, which is designed to increase effective memory and network bandwidth. Our experimental results show that this protocol can achieve significant improvements in running time performance (as much as a 4-fold improvement), as a result of the increase in effective bandwidth. In some cases, programs running under eager combining achieve better performance than the same program on a machine with 4 times more absolute bandwidth. We also compared eager combining to software broadcasting, and showed that eager combining consistently outperforms software broadcasting on our application programs.

Eager combining does not always help improve performance; in some cases, a program running on 128 processors runs slower than the same program on 64 processors. This is due to the fact that eager combining may still suffer from a lack of bandwidth in extreme cases of contention, especially when memory and network bandwidths are comparable. One way to further increase effective bandwidth for our applications is by using multi-level trees in the protocol. Multi-level trees introduce new complexities in the protocol, and require a delicate balance between increased latency and decreased contention. We intend to investigate these issues in detail, and extend our analysis of the protocol under these new assumptions.

References

[Agarwal, 1992] A. Agarwal. "Performance Tradeoffs in Multithreaded Processors," *IEEE Transactions on Parallel and Distributed Systems*, 3(5):525–539, Sept 1992.

[Agarwal *et al.*, 1992] A. Agarwal, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B.-H. Lim, G. Maa, and D. Nussbaum, "The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor," In M. Dubois and S. S. Thakkar, editors, *Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1992.

[Archibald and Baer, 1986] J. Archibald and J.-L. Baer, "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model," *ACM Transactions on Computer Systems*, 4(4):273–298, Nov 1986.

[Bennett *et al.*, 1990] J. K. Bennett, J. B. Carter, and W. Zwaenepoel, "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence," In *Proceedings of the 2nd ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 168–176, March 1990.

[Bianchini and Brown, 1993] R. Bianchini and C. M. Brown, "Parallel Genetic Algorithms on Distributed-Memory Architectures," In *Transputer: Research and Applications. NATUG-6*, pages 67–82, May 1993.

[Bianchini *et al.*, 1993] R. Bianchini, M. E. Crovella, L. Kontothanasis, and T. J. LeBlanc, "Alleviating Memory Contention in Matrix Computations on Large-Scale Shared-Memory Multiprocessors," Technical Report 449, Department of Computer Science, University of Rochester, April 1993.

[Carter *et al.*, 1991] J. B. Carter, J. K. Bennett, and W. Zwaenepoel, "Implementation and Performance of Munin," In *Proceedings of the 13th ACM Symposium on Operating System Principles*, pages 152–164, Oct 1991.

[Eggers and Katz, 1988] S. J. Eggers and R. H. Katz, "A Characterization of Sharing in Parallel Programs and its Application to Coherency Protocol Evaluation," In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 373–383, May 1988.

[Gallivan *et al.*, 1990] K. A. Gallivan, R. J. Plemmons, and A. H. Sameh, "Parallel Algorithms for Dense Linear Algebra Computations," *SIAM Review*, 32(1):54–135, March 1990.

[Glenn *et al.*, 1991] R. R. Glenn, D. V. Pryor, J. M. Conroy, and T. Johnson, "Characterizing Memory Hot Spots in a Shared-Memory MIMD Machine," *Proceedings of Supercomputing'91*, pages 554–566, November 1991.

[Gottlieb *et al.*, 1983] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer," *IEEE Transactions on Computers*, C-32(2):175–189, Feb 1983.

[Gupta and Weber, 1992] A. Gupta and W.-D. Weber, "Cache Invalidation Patterns in Shared-Memory Multiprocessors," *IEEE Transaction on Computers*, 41(7):794–810, July 1992.

[Kendall Square Research Corp., 1992] Kendall Square Research Corp., *KSR1 Principles of Operation*, Kendall Square Research Corporation, 170 Tracer Lane, Waltham, MA, 1992.

[Lenoski *et al.*, 1990] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, "The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor," *Proceedings of the 17th International Symposium on Computer Architecture*, pages 148–159, May 1990.

[Lenoski *et al.*, 1993] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy, "The DASH Prototype: Logic Overhead and Performance," *IEEE Transactions on Parallel and Distributed Systems*, 4(1):41–61, Jan 1993.

[Mellor-Crummey and Scott, 1991] J. M. Mellor-Crummey and M. L. Scott, "Synchronization Without Contention," *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 269–278, April 1991.

[MIPS Computer Systems Inc., 1991] MIPS Computer Systems Inc., *MIPS R4000 Microprocessor User's Manual*, Integrated Device Technology, Inc., 1991.

[Mowry and Gupta, 1991] T. Mowry and A. Gupta, "Tolerating Latency Through Software-Controlled Prefetching in Shared-Memory Multiprocessors," *Journal of Parallel and Distributed Computing*, 12(2):87–106, June 1991.

[Patel and Harrison, 1988] N. M. Patel and P. G. Harrison, "On Hot-Spot Contention in Interconnection Networks," *Performance Evaluation Review*, 16(1):114–123, May 1988, Originally published at SIGMETRICS '88.

[Pfister *et al.*, 1985] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss, "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture," In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 764–771, Aug 1985.

[Pfister and Norton, 1985] G. F. Pfister and V. Alan Norton, "'Hot Spot' Contention and Combining in Multistage Interconnection Networks," *IEEE Transactions on Computers*, C-34(10):943–948, October 1985.

[Rose, 1988] J. Rose, "LocusRoute: A Parallel Global Router for Standard Cells," In *Proceedings of the 25th Design and Automation Conference*, pages 189–195, June 1988.

[Rosti *et al.*, 1993] E. Rosti, E. Smirni, T.D. Wagner, A.W. Apon, and L.W. Dowdy, "The KSR1: Experience and Modelling of Poststore," In *Proceedings of the 1991 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, 1993.

[Veenstra, 1993] J. E. Veenstra, "Mint Tutorial and User Manual," Technical Report 452, Department of Computer Science, University of Rochester, July 1993.

[Weber and Gupta, 1989] W.-D. Weber and A. Gupta, "Analysis of Cache Invalidation Patterns in Multiprocessors," In *Proceedings of the 3rd Conference on Architectural Support for Programming Languages and Operating Systems*, pages 243–256, Apr 1989.

[Wilson Jr., 1987] Andrew W. Wilson Jr., "Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors," In *Proceedings of the 14th International Symposium on Computer Architecture*, pages 244–252, 1987.

[Wittie and Maples, 1989] Larry Wittie and Creve Maples, "MERLIN: Massively Parallel Heterogeneous Computing," In *Proceedings of the 1989 International Conference on Parallel Processing*, pages I–142 – I–150, August 1989.

[Yew *et al.*, 1987] Pen-Chung Yew, Nian-Feng Tzeng, and Duncan H. Lawrie, "Distributing Hot-Spot Addressing in Large-Scale Multiprocessors," *IEEE Transactions on Computers*, C-36(4):388–395, April 1987.